# Season 1: Casting Incompatible Types

## An Access Example

// Demonstrate casts.

class Conversion {

public

static void main(String[] args) {

byte b;

int i = 257;

double d = 323.142;

System.out.println("\nConversion of int to byte.");

b = (byte)i;

System.out.println("i and b " + i + " " + b);

System.out.println("\nConversion of double to int.");

i = (int)d;

System.out.println("d and i " + d + " " + i);

System.out.println("\nConversion of double to byte.");

b = (byte)d;

System.out.println("d and b " + d + " " + b);

}

}

This program generates the following output:

Conversion of int to byte.

i and b 257 1

Conversion of double to int.

d and i 323.142 323

Conversion of double to byte.

d and b 323.142 67

# Season 2: Using break like a goto

## An Access Example

**class Break {**

**public**

**static void main(String[] args) {**

**boolean t = true;**

**first : {**

**second : {**

**third : {**

**System.out.println("Before the break.");**

**if (t) break second; // break out of second block**

**System.out.println("This won't execute");**

**}**

**System.out.println("This won't execute");**

**}**

**System.out.println("This is after second block.");**

**}**

**}**

**}**

Running this program generates the following output:

Before the break.

This is after second block.

# Season 3: Class

## An Access Example

Here, b1 has been set to null, but b2 still points to the original object.

Box b1 = new Box();

Box b2 = b1;

// ...

b1 = null;

## Nested and Inner Classes

// Define an inner class within a for loop.

class Outer {

int outer\_x = 100;

void test() {

for (int i = 0; i < 10; i++) {

class Inner {

void display() {

System.out.println("display: outer\_x = " + outer\_x);

}

}

Inner inner = new Inner();

inner.display();

}

}

}

class InnerClassDemo {

public static void main(String[] args) {

Outer outer = new Outer();

outer.test();

}

}

The output from this version of the program is shown here:

display: outer\_x = 100

display: outer\_x = 100

display: outer\_x = 100

display: outer\_x = 100

display: outer\_x = 100

display: outer\_x = 100

display: outer\_x = 100

display: outer\_x = 100

display: outer\_x = 100

display: outer\_x = 100

# Season 4: Overloading

## Overloading Methods

class OverloadDemo {

void test() {

System.out.println("No parameters");

}

// Overload test for one integer parameter.

void test(int a) {

System.out.println("a: " + a);

}

void test(int a, int b) {

System.out.println("a and b: " + a + " " + b);

}

// Overload test for a double parameter

double test(double a) {

System.out.println("double a: " + a);

return a \* a;

}

}

## Overloading Constructor

class Box {

double width;

double height;

double depth;

// This is the constructor for Box.

Box(double w, double h, double d) {

width = w;

height = h;

depth = d;

}

}

## Using Objects as Parameters

// Objects are passed through their references.

class Test {

int a, b;

Test(int i, int j) {

a = i;

b = j;

}

// pass an object

void meth(Test o) {

o.a \*= 2;

o.b /= 2;

}

}

class PassObjRef {

public static void main(String[] args) {

Test ob = new Test(15, 20);

ob.meth(ob);

System.out.println("ob.a and ob.b after call: " + ob.a + " " + ob.b);

}

}

This program generates the following output:

ob.a and ob.b before call: 15 20

ob.a and ob.b after call: 30 10

# Season 5: Get, set, static

## Get, set

class Test {

private int c; // private access

// methods to access c

void setC(int i) { // set c's value

c = i;

}

int getC() { // get c's value

return c;

}

}

## Static

class StaticDemo {

static int a = 42;

static int b = 99;

static void callme() {

System.out.println("a = " + a);

}

}

class StaticByName {

public static void main(String[] args) {

StaticDemo.callme();

System.out.println("b = " + StaticDemo.b);

}

}

# Season 6: Using command-line arguments

## An Access Example

// Display all command-line arguments.

class CommandLine {

public static void main(String[] args) {

for (int i = 0; i < args.length; i++)

System.out.println("args[" + i + "]: " + args[i]);

}

}

Try executing this program, as shown here:

javac CommandLine.java

java CommandLine this is a test 100 -1

When you do, you will see the following output:

args[0]: this

args[1]: is

args[2]: a

args[3]: test

args[4]: 100

args[5]: -1

# Season 7: Varargs (Variable-Length Arguments)

Command structure

int doIt(int a, int b, double c, int ... vals) { //Correct!

int doIt(int a, int b, double c, int ... vals, boolean stopFlag) { // Error!

Code:

// Use varargs with standard arguments.

class VarArgs2 {

// Here, msg is a normal parameter and v is a

// varargs parameter.

static void vaTest(String msg, int ... v) {

System.out.print(msg + v.length +

" Contents: ");

for(int x : v)

System.out.print(x + " ");

System.out.println();

}

public static void main(String[] args)

{

vaTest("One vararg: ", 10);

vaTest("Three varargs: ", 1, 2, 3);

vaTest("No varargs: ");

}

}

The output from this program is shown here:

One vararg: 1 Contents: 10

Three varargs: 3 Contents: 1 2 3

No varargs: 0 Contents:

## Remember

*You can overload a method that takes a variable-length argument.*

*Somewhat unexpected errors can result when overloading a method that takes a variable-length argument. These errors involve* ***ambiguity*** *because it is possible to create an ambiguous*

*call to an overloaded varargs method.*

# Season 8: Inheritance

## When Constructors Are Executed

class A {

A() {

System.out.println("Inside A's constructor.");

}

}

// Create a subclass by extending class A.

class B extends A {

B() {

System.out.println("Inside B's constructor.");

}

}

// Create another subclass by extending B.

class C extends B {

C() {

System.out.println("Inside C's constructor.");

}

}

class CallingCons {

public static void main(String[] args) {

C c = new C();

}

}

The output from this program is shown here:

Inside A's constructor

Inside B's constructor

Inside C's constructor

## Using Abstract Classes

// A Simple demonstration of abstract.

abstract class A {

abstract void callme();

// concrete methods are still allowed in abstract classes

void callmetoo() {

System.out.println("This is a concrete method.");

}

}

class B extends A {

void callme() {

System.out.println("B's implementation of callme.");

}

}

class AbstractDemo {

public static void main(String[] args) {

B b = new B();

b.callme();

b.callmetoo();

}

}

## Using final to Prevent Overriding, Inheritance

class A {

final void meth() {

System.out.println("This is a final method.");

}

}

class B extends A {

void meth() { // ERROR! Can't override.

System.out.println("Illegal!");

}

}

final class A {

//...

}

// The following class is illegal.

class B extends A { // ERROR! Can't subclass A

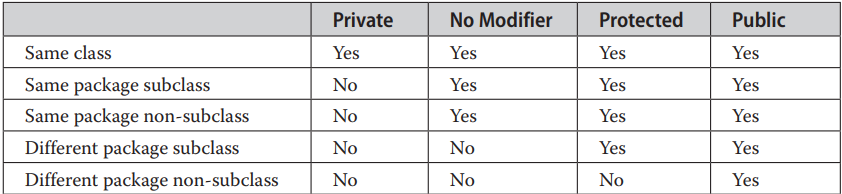
//...

}

## Remember

*Call to ‘super()’ must be first statement in constructor body*

# Season 9: Packages



## An Access Example

This is file Protection.java:

package p1;

public class Protection {

int n = 1;

private int n\_pri = 2;

protected int n\_pro = 3;

public int n\_pub = 4;

public Protection() {

System.out.println("base constructor");

System.out.println("n = " + n);

System.out.println("n\_pri = " + n\_pri);

System.out.println("n\_pro = " + n\_pro);

System.out.println("n\_pub = " + n\_pub);

}

}

This is file Derived.java:

package p1;

class Derived extends Protection {

Derived() {

System.out.println("derived constructor");

System.out.println("n = " + n);

// class only

// System.out.println("n\_pri = "4 + n\_pri);

System.out.println("n\_pro = " + n\_pro);

System.out.println("n\_pub = " + n\_pub);

}

}

This is file SamePackage.java:

package p1;

class SamePackage {

SamePackage() {

Protection p = new Protection();

System.out.println("same package constructor");

System.out.println("n = " + p.n);

// class only

// System.out.println("n\_pri = " + p.n\_pri);

System.out.println("n\_pro = " + p.n\_pro);

System.out.println("n\_pub = " + p.n\_pub);

}

}

This is file Protection2.java:

package p2;

class Protection2 extends p1.Protection {

Protection2() {

System.out.println("derived other package constructor");

// class or package only

// System.out.println("n = " + n);

// class only

// System.out.println("n\_pri = " + n\_pri);

System.out.println("n\_pro = " + n\_pro);

System.out.println("n\_pub = " + n\_pub);

}

}

This is file OtherPackage.java:

package p2;

class OtherPackage {

OtherPackage() {

p1.Protection p = new p1.Protection();

System.out.println("other package constructor");

// class or package only

// System.out.println("n = " + p.n);

// class only

// System.out.println("n\_pri = " + p.n\_pri);

// class, subclass or package only

// System.out.println("n\_pro = " + p.n\_pro);

System.out.println("n\_pub = " + p.n\_pub);

}

}

If you want to try these two packages, here are two test files you can use. The one for package p1 is shown here:

// Demo package p1.

package p1;

// Instantiate the various classes in p1.

public class Demo {

public static void main(String[] args) {

Protection ob1 = new Protection();

Derived ob2 = new Derived();

SamePackage ob3 = new SamePackage();

}

}

The test file for p2 is shown next:

// Demo package p2.

package p2;

// Instantiate the various classes in p2.

public class Demo {

public static void main(String[] args) {

Protection2 ob1 = new Protection2();

OtherPackage ob2 = new OtherPackage();

}

}

## Importing Packages

package mypack;

public class Balance {

String name;

double bal;

public Balance(String n, double b) {

name = n;

bal = b;

}

public void show() {

if(bal<0)

System.out.print("--> ");

System.out.println(name + ": $" + bal);

}

}

As you can see, the Balance class is now public. Also, its constructor and its show( ) method are public, too. This means that they can be accessed by any type of code outside the mypack package. For example, here TestBalance imports mypack and is then able to make use of the Balance class:

import mypack.\*;

class TestBalance {

public static void main(String[] args) {

/\* Because Balance is public, you may use Balance

class and call its constructor. \*/

Balance test = new Balance("J. J. Jaspers", 99.88);

test.show(); // you may also call show()

}

}

## Remember

*As an experiment, remove the public specifier from the Balance class and then try compiling TestBalance. As explained, errors will result.*

# Season 10: Interface

## An Access Example

interface Callback {

void callback(int param);

}

class Client implements Callback {

// Implement Callback's interface

public void callback(int p) {

System.out.println("callback called with " + p);

}

void nonIfaceMeth() {

System.out.println("Classes that implement interfaces " +

"may also define other members, too.");

}

}

class AnotherClient implements Callback {

// Implement Callback's interface

public void callback(int p) {

System.out.println("Another version of callback");

System.out.println("p squared is " + (p\*p));

}

}

class TestIface2 {

public static void main(String[] args) {

Callback c = new Client();

AnotherClient ob = new AnotherClient();

c.callback(42);

c = ob; // c now refers to AnotherClient object

c.callback(42);

}

}

## Nested Interfaces

// A nested interface example.

// This class contains a member interface.

class A {

// this is a nested interface

public interface NestedIF {

boolean isNotNegative(int x);

}

}

// B implements the nested interface.

class B implements A.NestedIF {

public boolean isNotNegative(int x) {

return x < 0 ? false: true;

}

}

class NestedIFDemo {

public static void main(String[] args) {

// use a nested interface reference

A.NestedIF nif = new B();

if(nif.isNotNegative(10))

System.out.println("10 is not negative");

if(nif.isNotNegative(-12))

System.out.println("this won't be displayed");

}

}

## Interfaces Can Be Extended

// One interface can extend another.

interface A {

void meth1();

void meth2();

}

// B now includes meth1() and meth2() -- it adds meth3().

interface B extends A {

void meth3();

}

// This class must implement all of A and B

class MyClass implements B {

public void meth1() {

System.out.println("Implement meth1().");

}

public void meth2() {

System.out.println("Implement meth2().");

}

public void meth3() {

System.out.println("Implement meth3().");

}

}

class IFExtend {

public static void main(String[] args) {

MyClass ob = new MyClass();

ob.meth1();

ob.meth2();

ob.meth3();

}

}

## Default Method Fundamentals

public interface MyIF {

// This is a "normal" interface method declaration.

// It does NOT define a default implementation.

int getNumber();

// This is a default method. Notice that it provides

// a default implementation.

default String getString() {

return "Default String";

}

}

// Implement MyIF.

class MyIFImp implements MyIF {

// Only getNumber() defined by MyIF needs to be implemented.

// getString() can be allowed to default.

Public int getNumber() {

return 100;

}

}

// Use the default method.

class DefaultMethodDemo {

public static void main(String[] args) {

MyIFImp obj = new MyIFImp();

// Can call getNumber(), because it is explicitly

// implemented by MyIFImp:

System.out.println(obj.getNumber());

// Can also call getString(), because of default

// implementation:

System.out.println(obj.getString());

}

}

The output is shown here:

100

Default String

## Use static Methods in an Interface

The following shows an example of a static method in an interface by adding one to MyIF, shown in the previous section. The static method is getDefaultNumber( ). It returns zero.

public interface MyIF {

// This is a "normal" interface method declaration.

// It does NOT define a default implementation.

int getNumber();

// This is a default method. Notice that it provides

// a default implementation.

default String getString() {

return "Default String";

}

// This is a static interface method.

static int getDefaultNumber() {

return 0;

}

}

The getDefaultNumber( ) method can be called, as shown here:

int defNum = MyIF.getDefaultNumber();

## REMEMBER

1. *When you implement an interface method, it must be declared as public.*
2. *If a class includes an interface but does not fully implement the methods required by that interface, then that class must be declared as abstract.*
3. *Static interface methods are not inherited by either an implementing class or a subinterface.*

# Season 11: Exceptions

This is the general form of an exception-handling block:  
try {  
// block of code to monitor for errors  
}  
catch (ExceptionType1 exOb) {  
// exception handler for ExceptionType1}  
catch (ExceptionType2 exOb) {  
// exception handler for ExceptionType2}  
// ...  
finally {  
// block of code to be executed after try block ends  
}

## Example